A Software Lab with On-demand Support

Daniel Dietsch
University of Freiburg
dietsch@informatik.uni-freiburg.de

Vincent Langenfeld
University of Freiburg
langenfv@informatik.uni-freiburg.de

Abstract

Software lab courses are in a paradoxical situation: teaching and experiencing software engineering processes requires large and complex projects as well as various tools but target students with extremely diverse prior knowledge, thus regularly overwhelming the more inexperienced ones. In this paper, we present a software lab course design: we expose teams of students to the full complexity of software development projects while simultaneously employing various systems that identify issues in teamwork or project progress and allow lecturers to provide on-demand support throughout all stages. The success of the course is shown by questionnaire data from the last ten years.

1. Introduction

For students, a software lab course can provide an environment in which they can first encounter complex software, the resulting need for software engineering practice, and the intricacies of working in a team that really depends on each other. Students can experience, often for the first time, how a development process allows them to tackle a problem that would be too complex for a single student inside the given timeframe, and which requires them to rely on their team members. In order to provide such an environment, it is often inevitable that students are at first overwhelmed by all the new areas they have to navigate. Often, a lab facilitates their first contact with a multitude of tools, like new IDEs, feature and issue trackers, version control systems with workflows (e.g., git flow), continuous integration, coding conventions and static analysis, new programming languages, new frameworks, and new problem domains. As if this is not enough, students are also confronted with social dynamics in the form of power struggles, social loafing [1], vastly different skill levels, or just plain scheduling conflicts.

We and others (e.g., [4, 3]) believe that overwhelming students in such a way is – to a certain degree – necessary to facilitate learning, but only if the course design can prevent students from despairing, provides them sufficient support at every stage to overcome these challenges, and allows them to set their own pace. In particular, we believe that overcoming these challenges as a team fosters a deep understanding of the usefulness of tools and techniques they are confronted with. The design of our software lab tries to achieve these goals.

We present in this paper our design of a software lab course, which is the result of refining a core concept over the last ten years by carefully surveying and measuring students' performance and satisfaction. The lab course is compulsory for the undergraduate computer science degree, and optional for various other MINT degrees at the University of Freiburg, usually as part of their third semester. Prior knowledge of students is rather varied: the majority of students following the syllabus have visited two programming lectures and no software engineering lecture, but may also be further along in their studies (see Table 1), or may have extensive previous knowledge.

We address the aforementioned problems with a set of core mechanisms: we assign students to groups based on a prior skill assessment, we use fictional customer requirements to steer projects to similar complexity, we use a modified version of Scrum together with various organizational and technical metrics to detect technical and organisational issues in groups, and we use admission criteria that can partly be controlled by the teams, which enables teams to take responsibility for the level of participation of team members and to self-organize more effectively.

Improving each groups' chance for success begins with group composition. Our basis for this composition is an extensive self-assessment questionnaire which has to be completed by the students in the first course lecture. Groups of equal size are formed such that at least one student with knowledge in each

URI: https://hdl.handle.net/10125/79450
978-0-9981331-5-7

necessary domain is present and students who score low on motivation are distributed equally over all teams as they have a much higher chance of leaving a team early, but also moderate too ambitious plans.

The project for all groups is given by a set of requirements for a computer game supplied by a fictional customer. We chose computer games for the project, as they are seen as fun and interesting by a large portion of the students [8] (compared to, e.g., business software) and have a simple success metric (the game is playable and does nothing uncalled-for). Computer games also have an inherent level of complexity that is high enough to allow students to experience the necessity of software engineering practices in the sense of a planned encounter to the first system [3]. Even students not interested in computer games can find engaging tasks as computer games touch nearly every area of computer science: for example, complex algorithms (for path planning, rendering, lighting, even custom sorting), data structures (optimized DS for spatial partitioning, efficient rendering, event handling, etc.), computer graphics, networking (for multiplayer), software engineering for an efficient and robust architecture, requirements analysis, testing, and of course debugging and programming itself are all part of the development of a computer game. As the course design tries to avoid influencing the feeling of being restrained by a larger architecture or a framework, which in turn would either limit the feeling of *owning* the project or having *impact*, i.e., non-efficacy on the resulting output or non-acceptance of the work necessary [6, 11], we require them to write their own game engine, using the MonoGame framework[1]. This also forces students to plan a sound software architecture from the ground up, a task usually unique to a lab course.

Throughout the lab, we use a modified version of Scrum as a process model. Sprints in the software lab are kept very short (one week), so that teams lose comparably little time if a sprint fails. Each group is supervised by a student teaching assistant (TA), who attends the *sprint meeting*, a singular meeting combining review, planning, and retrospective. In turn, all TAs meet with the lecturers in a weekly review meeting to discuss the current status of all teams and any issues that have to be addressed.

Over the whole software lab, the teaching staff (i.e., lecturers and TAs) is continuously available over several communication channels. The performance of groups, especially the lack thereof, is monitored per sprint meeting over metrics collected from a version control system (git with GitInspector), issue tracker (Gitea), build system (Jenkins), and static code analysis tools (Sonar, ReSharper). When problems with students or groups arise, TAs and lecturers decide on further actions based on our intervention plan, which is a growing collection of scenarios and responses that resulted in successful resolution in the past.

The remainder of the paper is organized as follows. After an introduction into the related work (Sec. 1.1) we give a general overview over a complete run of the course (Sec. 2) followed by a more detailed look at each of the different systems at work in the software lab. Next, we cover group composition (Sec. 2.1), the modified Scrum process (Sec. 2.2), and the requirements and design phase (Sec. 2.3). Finally, we present an overview of the support systems (Sec. 2.4), the grading scheme (Sec. 2.5), and some statistics and experiences of the last ten years of the course (Sec. 3).

1.1. Related Work

Many software lab courses target students of a higher semester where participants have attended at least one software engineering course. For example, 34 graduate and doctoral students [7], 27 students at the end of their Bachelors' education [9], and 97 students at the end of a three-year undergraduate program [2]. They do not explicitly describe their team formation process, so we assume it is left to the students.

Balaban and Sturm [2] report that many software engineering techniques are intended to help with large or complex projects whereas teaching is often done on small and easy to understand examples. In addition, most students perceive coding activities as much more rewarding than design and requirements analysis. Their suggested lab course is structured around a knowledgeable customer who writes good requirements and plans the development process. We agree with the initial assessment, but recommend a more realistic customer, who does not interfere with the development process and does not write good requirements (rather, writes ambiguous and incomplete ones). We offset the additional difficulty by using short development cycles, which allow the students to make mistakes but also enables them (and us) to realize and correct these mistakes.

Øystein Nytr et. al. [10] present an experiment on the degree of student choice in a lab course. Two conditions were explored in consecutive years of a large lab course. The first condition involved self-assigned teams freely choosing a project idea, pro-

[1] `www.monogame.net`

cess and technology. For the second condition teams were randomly assigned and a higher level of control was applied (e.g. project and planning were provided). Their experiment showed that involvement into freely chosen projects was higher and seen as more fun, but also increased the chance of team failure and was more prone to student lockout. Our design with an external group composition and requirements-driven individualization of the product can be seen as combination of both approaches: it inhibits effects of established groups while aiding in forming a new group identity around the product.

The lab course reported by Brügge et. al. [4] features the development of prototypes for projects presented by real customers from local businesses. Teams of students are composed according to several metrics including developing skill, and are then working according to an agile process model with an experienced team leader (i.e., an external project management) and a team coach (a student that already completed the course). The course design seems to give students a high degree of realism but is strongly dependent on contact to local businesses.

Aggarwal and O'Brian [1] conducted a survey on structural factors on social loafing by interviewing 420 university students' with past group project experience. They identified three significant factors: the project focus (i.e. duration and size of the project), the group size, and the lack of peer evaluation. If social loafing is present, grades based solely on the team effort are perceived as unfair by the students who actually contributed to the project. In our course, the maximum group size and duration are fixed by outside forces (semester size and allocated TAs, awarded ECTS[2] points.), and the project size is fixed by the intended project complexity. However, we employ several systems (e.g. individual time tracking and points) to give impactful peer evaluation and thus counteract social loafing.

2. Course Structure

In this section we give a chronological overview over the course following the schedule in Figure 1.

The lab course is designed to simulate a real-world software development project in one lecture period (usually 17 weeks) with a budget of 6 ECTS equivalent to an average semester work load of 180 hours. Within this time, students shall design (see Week 0 to 2 in Figure 1) and implement in weeks 2 to 17 a computer game according to a set of requirements of a fictional customer.

[2]European Credit Transfer and Accumulation System

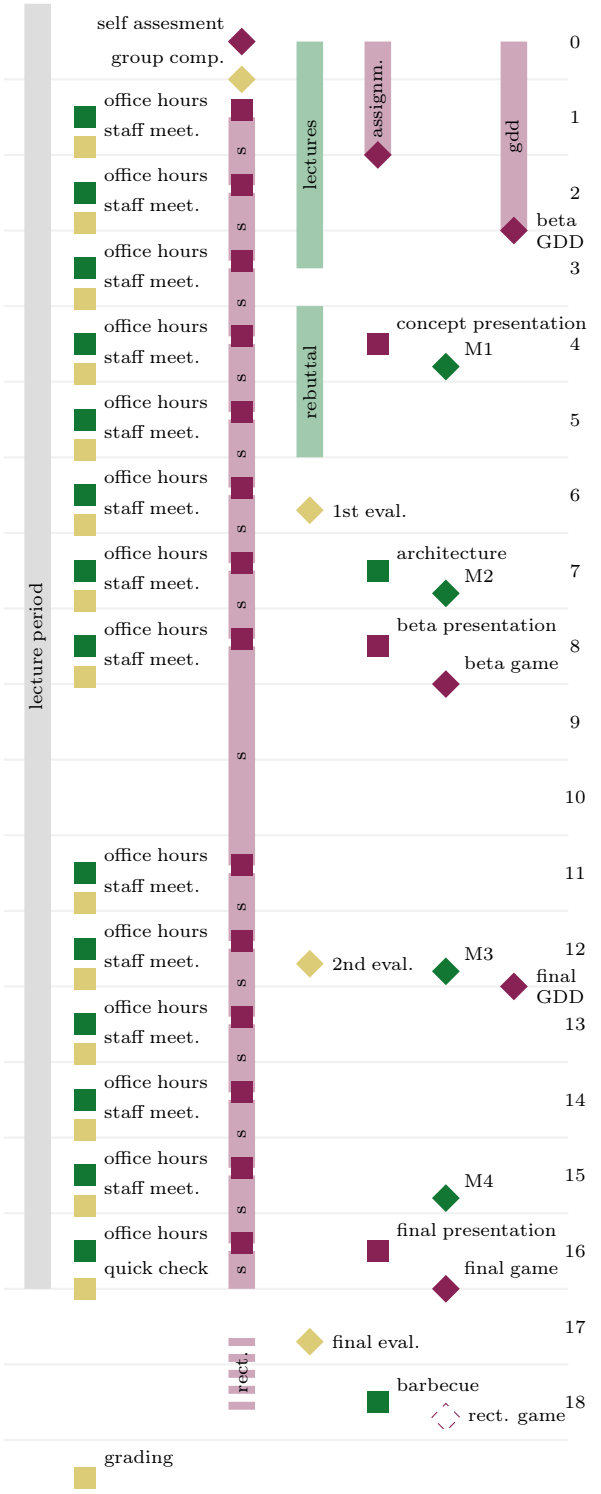

Figure 1. Layout of the software lab over the 17 weeks lecture period (with 9 and 10 being holidays). Bars depict longer running tasks, boxes events and diamonds artefacts. Responsibilities are marked by colours: students (wine), lecturers involved (green) or independent of students (sand). Dashed shapes depict optional events.

In the first week (see *group composition* in Figure 1) lecturers compose groups on the basis of a *self assessment* questionnaire to ensure each group has the abilities necessary to succeed. Students are also required to do an individual assignment within the fist two weeks (*assignm.*). This assignment is mainly to uncover problems preventing the students from working, i.e., opening tickets for all tasks of the assignment to get used to the issue tracker, committing name, email and the finished assignment to git to get used to the interaction with the source control system, and developing a small graphical program to ensure that the development environment is working. In the first four weeks lectures are held. Starting with an introduction into the course, the requirements, and the development process used in the lab course (week 0), game design and specification of game in a Game Design Document (GDD) (week 1), an overview on software engineering techniques in general (week 2) and software engineering specialized to game design (week 3). Thereby the lectures give a broad overview as a starting point for self-study. Each week the groups meet up with their TA in order to end the last sprint, and agree on tasks for the next sprint. Starting in week 1, lectures are also available for individual help in weekly office hours. Actual work on the project starts with a brief introduction in requirements engineering weeks 0 to 3 (*gdd*), in which the students design a game around set of fictive customer requirements. The result of this phase is written down in the form of a GDD and reviewed by the teaching staff in Week 2 (*beta GDD*). The GDD is used as the basis for the refinement of backlog items, i.e., user stories and tasks, for the remainder of the development in weeks 2 to 15 (*s* in Figure 1).

In week 4 the teaching staff finishes reviewing the GDDs and submits reviews to the groups' issue trackers, initiating the *rebuttal* period, during which all questions and misconceptions on the requirements are clarified. Students have to present the concept of their games in a plenary session *concept presentation*. Each presentation has a fixed time slot of 10 minutes, ending in a short Q&A session open for questions from teaching staff and all students in the course. As a scheduling aid for the students we suggest a number of milestones. In week 4 the first milestone (*M1*) is reached at which a movable object and camera as well as sounds and loading of a level should exist. In the *architecture* presentation (see Week 5 in Figure 1), groups meet with a lecturer and their TA to evaluate their planned software architecture to uncover missing elements and bad software design. Students are encouraged to prepare a component diagram and walk through, given a set of scenarios.

In week 6 the first evaluation is opened asking students for anonymous feedback on the lectures and their wellbeing. Milestone two is reached in week 7: the game should have several game objects, interactions, menus and a HUD. In a second plenary session (Week 8) the *beta presentation* each group presents their current progress (10 minutes presentation, five minutes Q&A). Subsequently, a beta version of the program is submitted, containing screenshots and documentation (e.g. cheat codes) in order to further emphasize the halfway point of the project, at which basic functions should work. After the winter break (see Week 11 in Figure 1), a second anonymous evaluation of the students' wellbeing is made. Also, milestone three is reached: enemy AI should work, all interactions and content should exist, as well as a first version of the tech demo. The *final GDD* has to be submitted, which is the basis for grading of feature completeness of the game. In the last week of the lecture period, each group presents their finished game in the plenary *final presentation* (15 minutes). The development ends with the submission of the *final game*. Usually within a day, lecturers check if all submissions pass the course, i.e., they are executable and pass a brief inspection wrt. feature-completeness. If this inspection fails, or other grave problems with the game are uncovered (e.g. instant crash or missing files), the group is allotted a *rectifying* period of usually one week resulting in the *rect. game* and a grade deduction.

Following the lecture period and after grades are announced, a barbecue is held for the whole course (i.e. teaching staff and students). The barbecue is an integral part of the lab course as it conveys a sense of closure and signals the lecturers regard for the effort invested by the students upfront.

2.1. Group Composition

Following the introductory lecture in week 0 (see Fig. 1), each student is required to fill in a self-assessment questionnaire. Based on the assessment, lecturers compose groups of up to seven students within the following two days. A TA is assigned to each group while avoiding possible conflicts e.g. group members and TA knowing each other.

We compose groups by calculating seven scores from the answers of each student. The scores represent a students' perceived ability in the following dimensions. **Programming** assesses the programming ability of a student independent of a specific programming language or domain. **Organisation** is

1. The item in Gitea is closed.
2. The item has an estimate and actual time spent.
3. All files relevant are checked in at the release branch.
4. The teaching assistant has acknowledged the successful completion of the item using the current state of the remote release branch.

Figure 2. The minimum DoD given to the students.

a students' experience with organizing a team as well as working in an organized team. **Gaming** assesses the students' experience with computer games and genres, with higher scores reflecting more knowledge about the genres exhibiting the characteristics requested by our requirements, thus providing each group with a domain expert. The measures **Sound** and **Graphics** asses the knowledge on tools for sound and graphics creation and programming with assets of that kind (e.g., shader programming). **Motivation** assesses the general motivation for the course. **Flags** is a binary assessment set by the lecturers. A flag is set if the student repeats the course or has given inconsistent answers in the various self-assessment questions. Each group is seeded with an expert (one of the highest scoring students in the questionnaire) for each dimension in order of their priority: coding, organization, gaming, graphics and sound. Students being flagged or rating low on motivation are also distributed equally between the groups to distribute potential free loaders and dropouts equally to mitigate their impact. The remaining students are then distributed and students are exchanged until all groups are balanced wrt. to the inter-group variance in each measure while still not violating the earlier constraints, and until all groups have a similar level of diversity wrt. semester, language, and course of study.

We use this scheme to give each group a comparable chance of success based on assigning each group at least one person being an 'expert' in each necessary skill, as well as preventing the formation of groups around existing cliques (i.e., as opposed to letting students organize themselves), which helps to avoid student lockout [10] together with the possibility of being an expert in some domain. Requests to be paired with other students are generally rejected.

Shared work then starts (at the end of week 0) with students receiving invitations to their teams' infrastructure and the introduction of their TA.

2.2. Modified Scrum

We use a modified version of Scrum as our process model. In the Scrum process model, software is developed incremental, in repeating development cycles (the so-called sprints). Sprints, usually of two to four weeks each, repeat activities like planning, coding and integration, leading to a finished product at the end of each sprint. A Scrum team has three roles: the *developer*, the *product owner* (PO), a member of the development team who guides the development, and the *scrum master*, who is responsible for helping with the process. During a sprint, developers work through a subset of items selected for the sprint from the list of pending items in the project (the *project backlog*). Between sprints, three meetings are held: one to end the last sprint (the *sprint review*), one to reflect on the last sprint (the *sprint retrospective*), and one to prepare the next sprint (the *sprint planning*). In the sprint planning, a set of items from the project backlog is selected to be worked on in the next sprint. The work load of items is estimated by the team to ensure a manageable workload. In the *sprint retrospective* the *product increment*, a version of the product including all the work done in the sprint is compared to the items from the sprint backlog, and their status of completion is decided. To govern this decision a set of rules, the *definition of done* is used. Between sprints, in the *sprint review* reflection on the process, development and team is used for improvement. Scrum suggests a short stand-up status meeting is held every day (the *daily scrum*).

In the software lab, the roles are distributed as follows. All students are developers. One student is the PO, but the role may be reassigned to another student in the sprint retrospective. Reassigning the PO role allows each student to try an organization role without a course-long commitment. Students in the PO role are told to invest approximately two hours (around one fourth of the usual weekly work load) into the item. The role of Scrum master is filled by the TA of each group, and lecturers sometimes take the role of customer.

In the lab course the length of sprints is fixed to one week. As the students work on a complex project in a mostly unknown domain, short sprints allow for short planning cycles. Thus mistakes made in a sprint do not cause a large setback, but only a singular week of development time to be lost.

In order to simplify scheduling all meetings are collapsed into one weekly two-hour meeting between sprints. It consists of a 45-minute sprint review in which the accomplished work is inspected

by the whole team, deciding if a backlog item is finished according the DoD and their understanding of the backlog item. We assume students to be hard-working, hence each student is awarded five points per sprint if they complete their assigned items. If an item is not finished according to the DoD, points are deducted. The points are used to calculate the individual part of the grade, and may cause a student to fail admission early if too many points are lost. To still foster willingness to attempt more complex items, students may also report that they did not manage to complete an item in advance to the sprint meeting. This report has to include a short explanation of the problems encountered and has to occur in a timely manner, but allows the student to return the item and prevents them for loosing points. A 15-minute sprint retrospective is held to give feedback on team and course, redistribute roles, as well as alter the DoD. The students can (and in our experience do) make additions to the DoD to improve the quality of their product or cooperation, especially if students show habits detrimental to the team's work. Teams are required to keep a minimum DoD (see Fig. 2), that prescribes that work is integrated and committed to the VCS as well as estimating and logging actual spent time for each item. This gives students feedback about the accuracy of their estimates and allows lecturers to intervene if a gap of time spent between team members forms. The logged time also allows us to adjust the workload for following years (see Fig. 4). A minimum of items with seven hours of estimated time finished on average in each sprint is also necessary in order to not lose admission. As the estimate for each item is made as part of sprint planning it is hard for free riders to avoid working while also improving students estimations. The last hour is used for sprint planning, i.e., selecting which items from the backlog should be processed in the next sprint and who works on which item. An important aspect of our sprint planning is estimating the work for each item based on a shared understanding of the required work.

We chose Scrum because it fosters self-organization and enforces a continuous integration of work into a shippable product as well as retrospection on the development process itself. These concepts feed directly into the course structure. Working self-organized by picking and accepting ones own challenges is a large factor in supporting motivation [11]. The short development cycles (which we shortened further to one week) are also used to introduce monitoring and safeguards by providing reliable and immediate feedback. In particular, immediate feedback improves engagement of the students [5, 11] and prevents students from getting stuck.

2.3. The Requirements Game

The projects accomplished in the lab course have to be comparable (of equal complexity and workload). We use a set of requirements to steer the (technical) complexity while leaving a large degree of freedom on the actual game design. This gives students the feeling of *owning* the product, which supports motivation and work efficacy [6, 11].

The requirements (see Fig. 3) are presented in the first lecture of the lab course, and framed as being provided by a fictional customer. The students are warned, that work with the customer will be necessary in order to understand the customers needs and that requirements may be rather vague and do not necessarily state their true intent.

Students describe their planned product in the format of a Game Design Document (GDD). In industry, content and extent of a GDD depend on the project stage and intended audience (e.g. developers, investors), but usually contain a concise description of all objects present in the game (*mechanics*), the interactions with and between the objects (*dynamics*), how the game shall look and feel (*aesthetics*) as well as the *experiences* intended for the player, an overview of the story of the game, and a technical overview (*assumptions and constraints*) [12]. In the software lab, a GDD structure is provided as guideline to the students. It is mainly focused on the game mechanics and dynamics as these directly translate to elements (and complexity) of the implementation. Students are required to give a brief overview of the aesthetics as this is necessary to fulfil e.g. Requirement 1. Inclusion of extended material as concept art and a story overview is encouraged to improve the feeling of a whole game, but not mandatory. For the software lab, the GDD has to include at least all information to evaluate if the requirements (see Fig. 3) are fulfilled. In context of questions regarding the requirements (not requirements engineering in general) lecturers try to mimic the negotiation with a real customer (this exercise is deepened in the software engineering course the subsequent semester [14] using the same requirements).

The GDD review process is modelled by usual academic conventions with a score system and multiple reviewers (usually a TA and at least one lecturer) giving constructive feedback. While playing the customer role for requirements questions

Functional requirements The game should have ...

1. 2D or 3D graphics (no ASCII)
2. sound effects and music
3. a minimum of two players, at least one of them human
4. real time
5. indirect controls (point & click)
6. a pause function
7. a menu (navigable entirely with the mouse input, except for keyboard inputs)
8. game objects
   (a) min. 5 controllable
   (b) min. 5 selectable
   (c) min. 5 non controllable of which at least three are collidable
   (d) min. 3 controllable, collidable and movable
9. at least 5 statistics
10. achievements
11. at least 1000 active game objects of type 8d possible at once (tech demo)
12. save and load should be possible any time, but not necessarily controlled by the player
13. min. 10 different actions
    (a) e.g. running or abilities
    (b) it must be possible, that each game object of type 8d is able to move from a point in the world to each other accessible point in the world without getting stuck or impeding each other excessively etc. (pathfinding)

Quality requirements

14. Develop a good product
15. Graphic quality is not relevant but it has to be consistent
16. Acoustic effects should be consistent

Environmental constraints

17. C# / F# with .NET Core 3.1
18. MonoGame 3.8
19. Executable on Windows 10 (x64)
20. Use Visual Studio Community 2019
21. No compiler and ReSharper warnings or errors (weekly basis)
22. No compiler errors

Figure 3. The fictional customer requirements presented to the students in the beginning of the course.

and in the subsequent rebuttal of the GDDs, teaching staff can gauge the complexity of the project and suggest changes. While students should improve their GDDs with the review feedback, changes that alter the scope of the game are only allowed with the agreement of the lecturers.

A ticket containing the reviews is opened for the rebuttal. Usually all questions stated in the reviews can be clarified by a short dialogue using the comment function of the issue tracker. In our experience students, although explicitly working with requirements the first time, are mostly successful in producing a sensible refinement of the requirements. To compensate for unforeseen events, the workload for a group can be reduced by dropping requirements e.g. 11, 10 and 9, but usually dropping features from the GDD is sufficient to compensate for e.g. a team member leaving.

2.4. Support Systems

We employ a set of additional safety nets to mitigate usual problems encountered in a software lab course.

Students usually have no prior experience in planning larger software projects. The milestones in Figure 1 (M1 to M4) give a reference schedule for implementing features necessary to realize the requirements. Students are encouraged to modify the milestones according to their product. In the weekly teaching staff meeting, these milestones are used to evaluate each groups progress. Based on the TAs observations, metrics (e.g. git stats, students commits and tickets) and the status of the milestones, we try to predict if a group will reach the next milestone in time and the overall project goals. We abstract this prediction on a traffic light scale. Green means the group performs well and we predict they will reach their goals, yellow means we are unsure if they can reach the overall goal, but it is not necessary to intervene, and red means we believe the group will not reach their goals and/or intervention is necessary. Intervention takes the form of extra team meetings with lecturers that address the main obstacles faced by the group.

Students are required to acquire most of the information necessary for their project by themselves. We provide a head-start on our course wiki, which gives general information on the course organization, as well as primers for various game development topics. If no answer is found, students are encouraged to ask their team members, including their TAs. We provide several means of communication for groups, such as a group chat, the ticketing system and email and a course wide forum. TAs are instructed to support the information gathering process and not outright solve the problem of the students. Lecturers are also available in weekly office hours (see Fig. 1) and at nearly any time on the chat system. For the office hours lecturers go to the student pool to signal that they are open for questions, joint problem-solving, and the odd conversation.

The course imposes a number of admission criteria on students (5 points per sprint, average estimated time, compulsory attendance in team meetings and presentations, continuous contributions) that are mainly used to prevent social loafing and to foster team cohesion. As these criteria

are sometimes difficult to track, we provide a personal dashboard to each student that informs them in real-time about their current state and provides additional metrics about their personal contributions as well as an anonymized comparison with all other course members.

2.5. Grading

The final grade of the software lab is based on the individually reached points (see Sec. 2.2) and the final grade of the game. Both have to be passing grades to pass the course. As groups have to make decisions on the shape of their work and final product, both grading schemes are presented in the initial lecture.

To measure the individual engagement, each student is awarded five points per sprint (including the homework), if all assigned backlog items were finished according to the DoD or handed back (see Sec. 2.2). The individual grade is calculated by distributing the grading steps over the range of 0 to 19 lost points, i.e., a student losing 20 points automatically fails and is removed immediately. This may seem harsh, but students successfully finishing the lab in 19/20 as well as 20/21 averaged 67.7 of 70 points. Since their introduction, no student failed the course because of the individual points.

Grading of the final product is done by a rating on a set of dimensions (the FAUST scale): For *features* we rate the degree to which all features in the final GDD are implemented is (weighted triple). For *artefacts* we rate the quality of the overall submitted artefacts (GDD, CI build failures, Sonar warnings). For *usability* we rate the usability of the game according to a checklist (weighted double). For *sport* we rate how much fun the game made (weighted double). As fun is hard to measure, we rate the games on how well they are designed from the perspective of the game domain (e.g. using [13]). A taxonomy over a course's games is refined to discuss the relation of *a game being more fun than another* by the teaching staff, until consensus is reached. For *Tech Demo* we rate how well the tech demo of the game works (showing at least 1000 active units, see Fig. 3). These dimensions also communicate (especially to the product owner) how different features should be prioritized.

The final grade for the game is calculated by taking the weighted average of the dimensions rounded up. Positive or negative modifiers are then applied to compensate for unforeseen circumstances (e.g., loosing too many team members, using the rectifying period). Thus, even if the game is bad, the resulting individual grade can be considerably better if the student was supporting the team. Conversely, if a student managed to evade violating the admission criteria but still engaged poorly with the team, the student will receive a bad individual grade and cannot benefit from the work of the team.

Course	Part.	Drop.	Avg. TS	Avg. sem.	% Maj. CS
2013	73	10	7.3	4.29	95.9
2014	53	5	5.3	4.45	92.5
2015	49	4	4.9	4.20	100.0
2016	62	2	6.2	5.10	90.3
2017	68	5	6.8	4.56	97.1
2018	85	9	7.1	4.53	94.1
2019	73	12	7.0	5.15	95.9
2019/2020	101	16	7.2	3.87	97.0
2020/2021	96	15	6.7	4.31	93.8

Table 1. Key data for the last nine lab instances. Columns show course size (Part.), students that dropped out (Drop.), average team size (Avg. TS), average semester (Avg. sem.), and percentage of computer science majors.

3. Experience

Over the last nine instances of our course, we collected various metrics and adjusted our syllabus repeatedly. Some of these adjustments were driven due to external changes. For example, in 2013 we were suddenly confronted with 50% more students. Because we could not accommodate more teams, we increased the team size from 5 to 7 students, which would later become the norm. With this increase we introduced the requirement for a tech demo (see 11 in Fig. 3) and various additional roles in the teams. As larger teams lead to an increase in social loafing, we discovered that in many teams, only a small subset of the team members performed most of the actual work. This is expected in our team composition, as each team has members that are not as able or as committed to the project. Nevertheless, it is a common and understandable source of dissatisfaction among the students [1] (cf. Fig. 5, "I was dissatisfied…"). As a consequence, we implemented additional admission criteria, namely being able to influence the DoD in 2015 (i.e. enabling students to influence their teamwork and quality goals) and mandatory time tracking in 2019/2020. The aggregation of statistics in the personal dashboard (starting from 2021) to ease dealing with the admission criteria was received positively. Although the bar for admission is still rather low, a minimum of consistent contribution is enforced by making non-contribution significantly harder, e.g., by forcing students to work in order to avoid contributing. These measures improved the perceived social

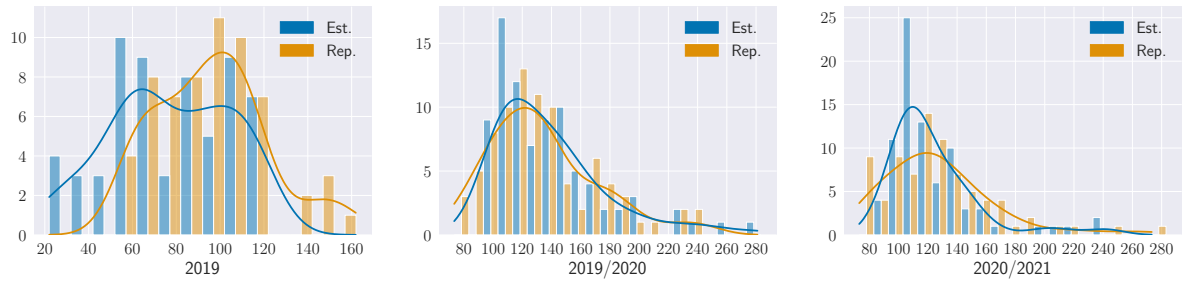

Figure 4. Histograms for estimated (Est.) and reported (Rep.) time (x-axis) for students (y-axis) over the last three years.

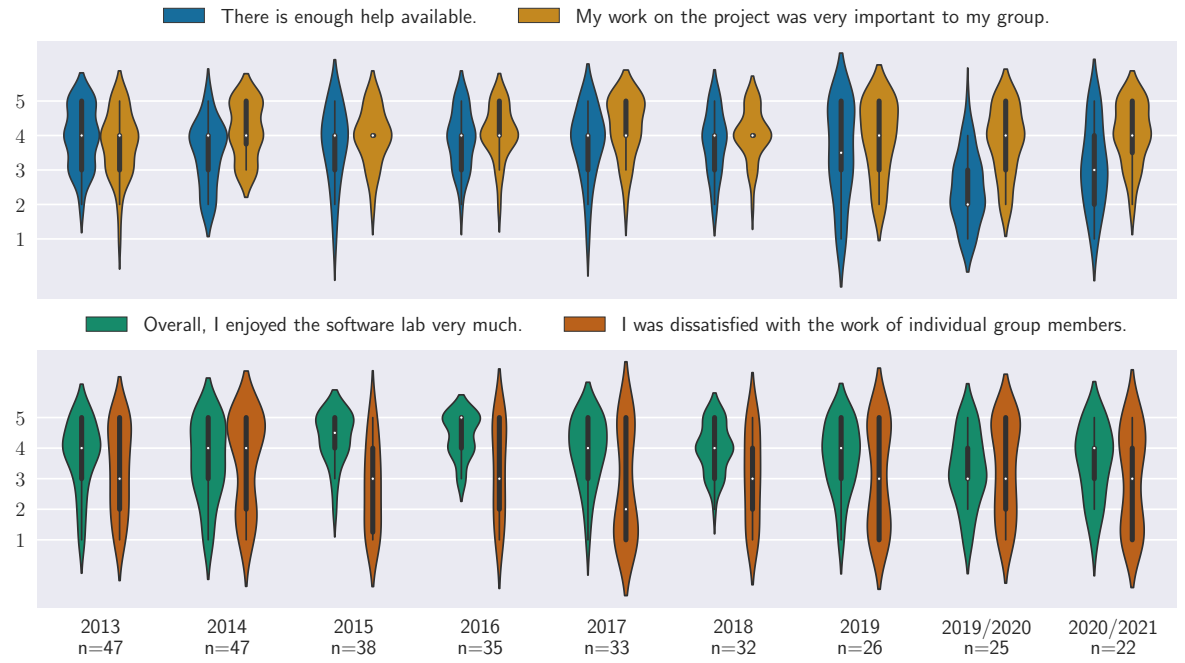

Figure 5. Final evaluation results concerning the work and group experience over the last 9 course instances. Violin plots show the distribution of the answers, the thick black bar in the middle shows the interquartile range, the white dot is the median, the x-axis is labelled with the course term, and the y-axis on a Likert scale ranging from "strongly disagree" (1) to "strongly agree" (5).

loafing problem substantially, while the number of dropouts increased only slightly (cf. Tab. 1).

The introduction of the additional task-size admission criterium had an interesting side effect: students were suddenly much more interested in quantifying the size of tasks, which lead to an improved accuracy not only in estimating tasks, but also in reporting time spent on tasks. This uncovered that students spent more time for the lab course than previously measured (cf. Fig. 4). In 2019, before the timed admission criterion, only 10% of our participants reported more than 132h of work (of the expected 120 to 130 hours). In 2019/2020, 44% crossed the threshold. As a consequence, we restructured our intervention guidelines s.t. we intervene earlier if we detect a project spending too much time on certain aspects of the game, and we started emphasizing the goal of staying inside the amount of work during the initial lecture. In 2020/2021 a higher number of students clustered around the expected work load with a remaining 26.8% reporting over 132h.

In order to gauge satisfaction and to detect and mitigate potential problems and misunderstandings, we perform three anonymous online evaluations. Two of them (1st and 2nd eval. in Fig. 1) are rather short regarding performance and quality of lecturers and TAs, and three free text questions for any problems with the course, IT infrastructure, and the questionnaire. The third questionnaire (final eval. in Fig. 1) is more extensive (around 100 questions

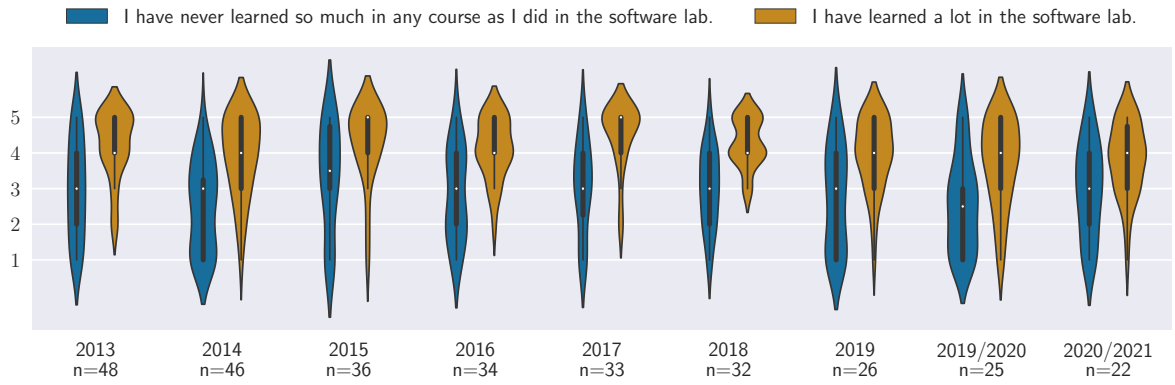

Figure 6. Final evaluation results concerning the overall learning experience over the last 9 course instances.

taking approx. 30 min) and is an important retrospective tool for the syllabus improvement cycle. Figures 5 and 6 report some overview results from the final questionnaire (Questions are stated on a five point Likert scale from (1) "strongly disagree" to (5) "strongly agree"). Figure 5 shows that students perceive their work as important for the group (as intended to foster motivation). Unfortunately the perception of help being offered did sink strongly around 2019/2020. We hypothesize that this is due to additional and more complicated tools being introduced to the course (e.g. Git replacing SVN), while neglecting additional course material regarding those tools, especially in the introductory lectures as well as involvement of the lecturers in other time-consuming projects. Additional communication tools (e.g. Mattermost and Discourse) and reinforced lecturer involvement did alleviate this in 2020/2021, although the course had to be held entirely virtual due to the COVID-19 pandemic. While social loafing is getting more difficult, it is sometimes supported by groups in order to avoid confrontation. We are aware of this behaviour and motivate groups to use the systems given to them (e.g. the DoD) to intervene. Nonetheless, there are groups in which students are dissatisfied with other students' work (see Fig. 5). We continue to investigate ways to improve our group composition schema and our intervention procedures to alleviate this issue.

Overall students consistently report that they learned a lot, and that (except 2014 and 2019/2020) they are learning more than in any other course.

References

[1] P. Aggarwal and C. L. O'Brien. Social loafing on group projects: Structural antecedents and effect on student satisfaction. *Journal of Marketing Education*, 30(3):255–264, 2008.

[2] M. Balaban and A. Sturm. Software Engineering Lab: An Essential Component of a Software Engineering Curriculum. In *ICSE (SEET)*, pages 21–30. ACM, 2018.

[3] F. P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering*. Pearson Education, 1995.

[4] B. Bruegge, S. Krusche, and L. Alperowitz. Software engineering project courses with industrial clients. *ACM Trans. Comput. Educ.*, 15(4):17:1–17:31, 2015.

[5] R. Chatley and T. Field. Lean Learning - Applying Lean Techniques to Improve Software Engineering Education. In *ICSE-SEET*, pages 117–126. IEEE Computer Society, 2017.

[6] J. R. Hackman and G. R. Oldham. Motivation through the Design of Work: Test of a Theory. *OBHDP*, 16(2):250–279, 1976.

[7] H. Hsu. Practicing Scrum in Institute Course. In *HICSS*, pages 1–9. ScholarSpace, 2019.

[8] M. Kosa, M. Yilmaz, R. V. O'Connor, and P. M. Clarke. Software engineering education and games: A systematic literature review. *J. Univers. Comput. Sci.*, 22(12):1558–1574, 2016.

[9] M. Kropp and A. Meier. Teaching Agile Software Development at University Level: Values, Management, and Craftsmanship. In *CSEE&T*, pages 179–188. IEEE, 2013.

[10] Ø. Nytr, A. Nguyen-Duc, H. Trætteberg, M. Lorås, and B. A. Farschian. Unreined Students or Not: Modes of Freedom in a Project-Based Software Engineering Course. In *CSEE&T*. IEEE, 2020.

[11] J. Reeve. A self-determination Theory Perspective on Student Engagement. In *Handbook of Research on Student Engagement*, pages 149–172. Springer, 2012.

[12] M. G. Salazar, H. A. Mitre-Hernández, C. L. Olalde, and J. L. G. Sánchez. Proposal of Game Design Document from Software Engineering Requirements Perspective. In *CGAMES*, pages 81–85. IEEE Computer Society, 2012.

[13] J. Schell. *The Art of Game Design: A Book of Lenses*. CRC press, 2008.

[14] B. Westphal. An Undergraduate Requirements Engineering Curriculum with Formal Methods. In *REET@RE*. IEEE Computer Society, 2018.